EDA-Laboratorio 2

28/10/2013

Mikel Barcina
Jose Ángel Gumiel

Índice de contenido

1 Introducción

En este laboratorio tenemos que trabajar con la estructura de listas doblemente ligadas. Se nos pide que seamos capaces de diseñar algoritmos que permitan añadir nuevos elementos de tipo genérico a las listas de diferente forma (al inicio, al final, después de un elemento concreto, ordenado o desordenado). Además también tenemos que implementar otras funciones típicas de listas a esta estructura, como pueden ser eliminar elementos, hallar el tamaño de la lista, buscar y tomar elementos, comprobar si está vacía, etc.

Por último tenemos que aplicar esta estructura en una lista del primer laboratorio, y comprobar que funciona correctamente.

La práctica ha de contener también los casos de prueba que hemos usado para comprobar la veracidad de nuestra implementación.

2 Diseño de las clases

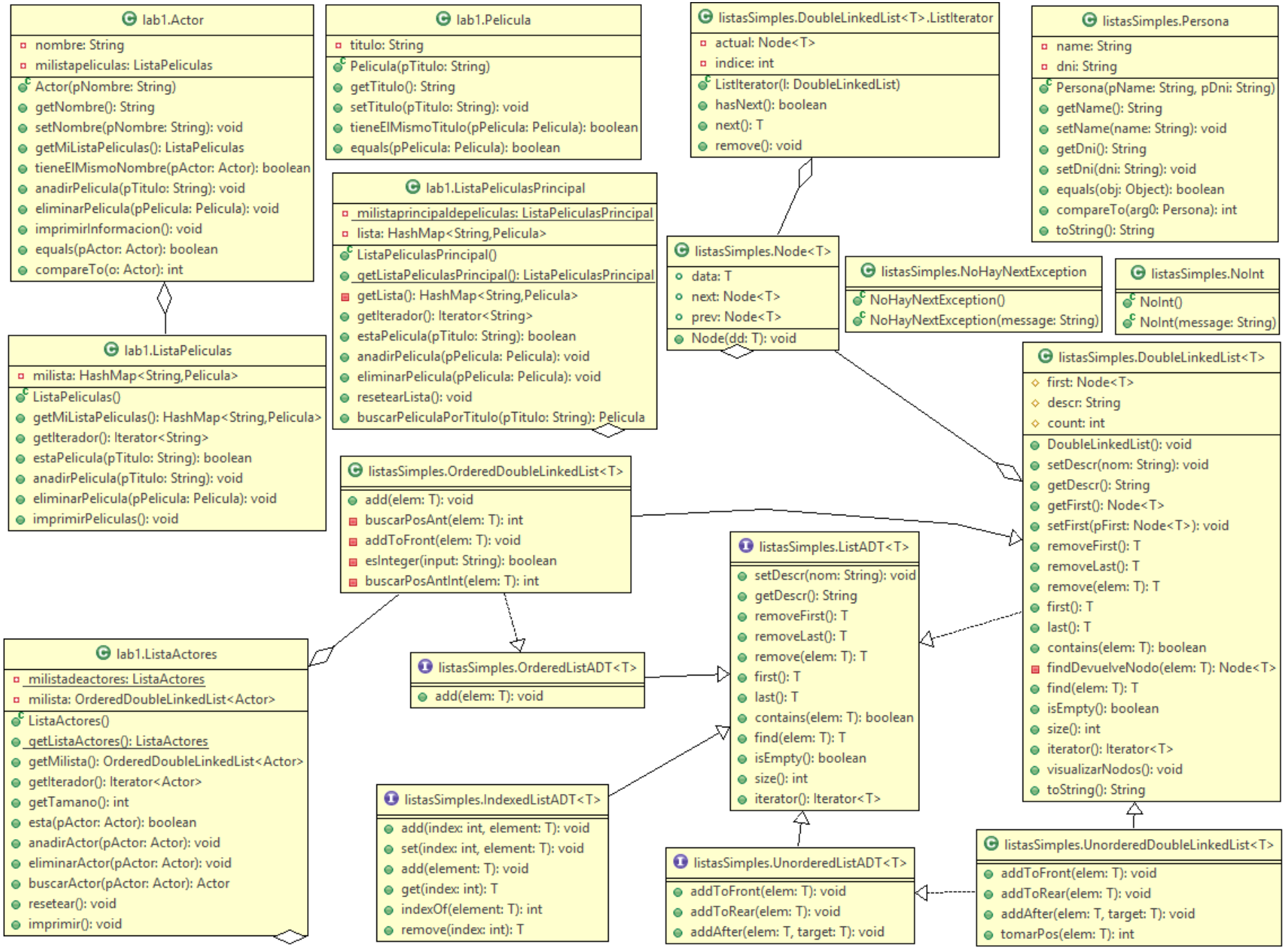
Como hemos venido haciendo hasta ahora, todas las clases que empleamos tienen los atributos privados y son accesibles mediante getters y setters. Así mismo, también serán privados los métodos que sólo vayan a ser utilizados por una única clase.

En el paquete listasSimples tenemos 11 clases, de las que destacamos las siguientes:

- DoubleLinkedList: Esta clase es la que define la estructura de datos de tipo lista doblemente ligada. Contiene todos los métodos que nos permiten trabajar con ella. Contains, find, isEmpty, size, remove, getFirst o visualizarNodos son algunos de los más significativos.
- Node: Es la clase que define lo que es el nodo. El nodo almacena información en su interior, contiene un dato que es de tipo T y apunta al siguiente elemento y a su antecesor.
- OrderedDoubleLinkedList: Es la clase lista ordenada doblemente ligada. Esta clase contiene el método add, que lo que hace es añadir en orden. Para que funcione correctamente hemos diseñado algunos métodos privados, que os facilitan la tarea. Hemos implementado un addToFront(), para el caso en el que el primer elemento de la lista sea mayor que el primero que se quiere insertar. El método buscarPosAnt() nos da la posición en la que debemos de insertar el elemento

  Tuvimos que hacer un caso especial para los números, ya que este método de ordenación no nos servía, dado que en el caso de ordenar las cifras [20, 1, 3] lo hacía como [1, 20, 3]. Esto era porque se tomaba el primer elemento para la ordenación, y el carácter “2” del número “20” está alfabéticamente antes que el “3”.
- UnorderedDoubleLinkedList: Muy similar a la anterior, sólo que el contenido de los elementos no es relevante para su ordenación. Disponemos de los métodos addToFront(), addToRear() y addAfter(), para este último tuvimos que diseñar el método privado tomarPos(). Este método toma la posición del elemento que buscamos, y nos da una referencia para saber dónde insertar el nuevo elemento.
- Clases adicionales: Tenemos dos clases que utilizamos para definir dos excepciones diferentes y la clase Persona.

En el siguiente diagrama de clases se pueden observar todas las clases mencionadas anteriormente y la totalidad de sus métodos y atributos, así como la relación existente entre ellas.


lab1.Actor
nombre: String
milistapeliculas: ListaPeliculas
Actor(pNombre: String)
getNombre(): String
setNombre(pNombre: String): void
getMiListaPeliculas(): ListaPeliculas
tieneElMismoNombre(pActor: Actor): boolean
anadirPelicula(pTitulo: String): void
eliminarPelicula(pPelicula: Pelicula): void
imprimirInformacion(): void
equals(pActor: Actor): boolean
compareTo(o: Actor): int
lab1.Pelicula
titulo: String
Pelicula(pTitulo: String)
getTitulo(): String
setTitulo(pTitulo: String): void
tieneElMismoTitulo(pPelicula: Pelicula): boolean
equals(pPelicula: Pelicula): boolean
listasSimples.DoubleLinkedList<T>.ListIterator
actual: Node<T>
indice: int
ListIterator(l: DoubleLinkedList)
hasNext(): boolean
next(): T
remove(): void
listasSimples.Persona
name: String
dni: String
Persona(pName: String, pDni: String)
getName(): String
setName(name: String): void
getDni(): String
setDni(dni: String): void
equals(obj: Object): boolean
compareTo(arg0: Persona): int
toString(): String
lab1.ListaPeliculasPrincipal
milistaprincipaldepeliculas: ListaPeliculasPrincipal
lista: HashMap<String,Pelicula>
ListaPeliculasPrincipal()
getListaPeliculasPrincipal(): ListaPeliculasPrincipal
getLista(): HashMap<String,Pelicula>
getIterador(): Iterator<String>
estaPelicula(pTitulo: String): boolean
anadirPelicula(pPelicula: Pelicula): void
eliminarPelicula(pPelicula: Pelicula): void
resetearLista(): void
buscarPeliculaPorTitulo(pTitulo: String): Pelicula
listasSimples.Node<T>
data: T
next: Node<T>
prev: Node<T>
Node(dd: T): void
listasSimples.NoHayNextException
NoHayNextException()
NoHayNextException(message: String)
listasSimples.NoInt
NoInt()
NoInt(message: String)
lab1.ListaPeliculas
milista: HashMap<String,Pelicula>
ListaPeliculas()
getMiListaPeliculas(): HashMap<String,Pelicula>
getIterador(): Iterator<String>
estaPelicula(pTitulo: String): boolean
anadirPelicula(pTitulo: String): void
eliminarPelicula(pPelicula: Pelicula): void
imprimirPeliculas(): void
listasSimples.DoubleLinkedList<T>
first: Node<T>
descr: String
count: int
DoubleLinkedList(): void
setDescr(nom: String): void
getDescr(): String
getFirst(): Node<T>
setFirst(pFirst: Node<T>): void
removeFirst(): T
removeLast(): T
remove(elem: T): T
first(): T
last(): T
contains(elem: T): boolean
findDevuelveNodo(elem: T): Node<T>
find(elem: T): T
isEmpty(): boolean
size(): int
iterator(): Iterator<T>
visualizarNodos(): void
toString(): String
listasSimples.OrderedDoubleLinkedList<T>
add(elem: T): void
buscarPosAnt(elem: T): int
addToFront(elem: T): void
esInteger(input: String): boolean
buscarPosAntInt(elem: T): int
listasSimples.ListADT<T>
setDescr(nom: String): void
getDescr(): String
removeFirst(): T
removeLast(): T
remove(elem: T): T
first(): T
last(): T
contains(elem: T): boolean
find(elem: T): T
isEmpty(): boolean
size(): int
iterator(): Iterator<T>
lab1.ListaActores
milistadeactores: ListaActores
milista: OrderedDoubleLinkedList<Actor>
ListaActores()
getListaActores(): ListaActores
getMilista(): OrderedDoubleLinkedList<Actor>
getIterador(): Iterator<Actor>
getTamano(): int
esta(pActor: Actor): boolean
anadirActor(pActor: Actor): void
eliminarActor(pActor: Actor): void
buscarActor(pActor: Actor): Actor
resetear(): void
imprimir(): void
listasSimples.OrderedListADT<T>
add(elem: T): void
listasSimples.IndexedListADT<T>
add(index: int, element: T): void
set(index: int, element: T): void
add(element: T): void
get(index: int): T
indexOf(element: T): int
remove(index: int): T
listasSimples.UnorderedListADT<T>
addToFront(elem: T): void
addToRear(elem: T): void
addAfter(elem: T, target: T): void
listasSimples.UnorderedDoubleLinkedList<T>
addToFront(elem: T): void
addToRear(elem: T): void
addAfter(elem: T, target: T): void
tomarPos(elem: T): int

3 Descripción de las estructuras de datos principales

Para este laboratorio hemos usado la estructura de listas doblemente ligadas o double linked lists:

- Lista doblemente ligada: Es un tipo de lista enlazada más completo. Cada nodo tiene dos enlaces: uno que apunta al nodo anterior y otro que apunta al nodo siguiente. En este caso vamos más allá, puesto que también es circular, es decir, que el último elemento apunta al primero, y el primero apunta al último.

4 Diseño e implementación de los métodos principales

Para presentar los métodos principales los agruparemos por Clases. Sólo nos centraremos en los métodos que consideremos más relevantes.

Clase DoubleLinkedList:

- Método contains()

  Se le pasa un elemento de tipo T como parámetro y nos da como resultado un booleano. Sirve para indicar si dicho elemento se encuentra en la lista o no.

  Casos de prueba:

  - Buscar un elemento que esté y nos devuelva true.
  - Buscar un elemento que no esté y nos devuelva false.

  Coste:

  - El coste es de orden n, O(n). En el peor de los casos tendremos que recorrer la lista por completo.

- Método find()

  Es muy similar al anterior, sólo que en vez de un booleano nos devuelve el elemento que estamos buscando.

  Casos de prueba:

  - Los mismos que en el ejemplo anterior.

  Coste:

  - El coste es de orden n, O(n). En el peor de los casos tendremos que recorrer la lista por completo.

- Método remove()

  Dado un elemento de tipo T, este es buscado en la lista, y si se encuentra será eliminado. Tendremos que fijarnos en que al proceder con el borrado los nodos apunten correctamente al siguiente elemento de la cadena

  Casos de prueba:

  - Eliminar un elemento que sabemos que está en la lista.
  - Eliminar un elemento que no se encuentra en la lista.

  Coste:

  - El coste es de orden n, O(n). En el peor de los casos tendremos que recorrer la lista por completo.

  Debemos de añadir que existen los métodos removeLast() y removeFirst(), que se encargan de borrar el último o el primer elemento. Estos tienen un coste constante O(1).

- Método visualizarNodos()

  Permite mostrar el contenido del nodo por pantalla

- ListIterator

  Es una clase que se encuentra dentro de la clase DoubleLinkedList, se trata de un iterador que hemos tenido que implementar.

Clase OrderedDoubleLinkedList:

- Método add()

  Es el método principal, añade un elemento de tipo T en orden a la lista, para ello hemos necesitado utilizar otros subprogramas. Para el orden convertimos el data en una cadena de caracteres. El coste en el peor de los casos será de O(n).

- Método addToFront()

  Añade un elemento al principio de la lista. Es el mismo que utilizamos en la clase "UnorderedDoubleLinkedList". Lo empleamos en el caso de que el primer elemento sea mayor que el elemento que se quiere insertar.

- Método buscarPosAnt()

  Recorre la lista hasta que encuentra la posición en la que hay que introducir el nuevo elemento, nos devuelve un integer. Con esta información, el método add sabe en qué posición hay que insertar.

Cuando tratamos con números hemos tenido problemas para ordenar, es por eso que lo hemos separada como un caso excepcional

- Método esInteger()

  Nos dice si el contenido del nodo es un integer, para ello intentamos transformar una cadena de caracteres en un integer. Si eso es posible devolvemos true, en caso contrario capturamos la excepción y devolvemos un false.

- Método buscarPosAntInt()

  Es igual que el método buscarPosAnt() pero diseñado para trabajar con números.

Clase UnorderedLinkedList:

- Método addAfter()

  Recibe dos elementos, uno que no está en la lista y otro que sí que está. Busca la posición del elemento y el nuevo se inserta en la posición inmediatamente después a la de este.

  Para que funcione correctamente recurrimos al método privado tomarPos, que busca la posición para que podamos recorrer posteriormente la lista con facilidad.

  Casos de prueba:

  - Que el elemento esté en la lista.
  - Que el elemento no se encuentra en la lista.
  - Que se añada correctamente después del elemento

  Coste:

  - El coste en el peor de los casos sería 2n, por lo que decimos que es O(n).

5 Código

5.1 Programa

A continuación mostramos la totalidad del código que hemos implementado, ordenado por clases:

5.1.1 Clase DoubleLinkedList

```
package listasSimples;
import java.util.Iterator;
public class DoubleLinkedList<T> implements ListADT<T> {

	// Atributos
	protected Node<T> first; // apuntador al primero
	protected String descr;  // descripciOn
	protected int count;

	// Constructor
	public DoubleLinkedList() {
		first = null;
		descr = "Esta es una lista circular";
		count = 0;
	}

	//Getters y Setters
	public void setDescr(String nom) {
	  descr = nom;
	}

	public String getDescr() {
	  return descr;
	}

	public Node<T> getFirst(){
		return first;
	}

	public void setFirst(Node<T> pFirst){
		this.first = pFirst;
	}



	public T removeFirst() {
	//Elimina el primer elemento de la lista

	    T resultado = null;
```

```
        if (!isEmpty()){
             Node<T> result = first;
             if (first == first.next){
                //Solo hay un elemento, luego despues de borrar la lista esta vacia
                first = null;
            }else {
                //Hay mas de un elemento
                Node<T> anterior = result.prev;
                Node<T> siguiente = result.next;
                first = siguiente;
                first.prev = anterior;
                result.prev.next = first;
            }
        count--;
        resultado = result.data;
        }
        return resultado;
    }


    public T removeLast() {
    //Elimina el ultimo elemento de la lista
            Node<T> rdo = null;
            if(!isEmpty()){
                rdo = first.prev;
                if(first==rdo){
                    //Solo hay un elemento en la lista que es el que quiero borrar
                    first = null;
                }else{
                    //Hay mas de un elemento
                    Node<T> anterior = rdo.prev;
                    Node<T> siguiente = first;
                    anterior.next = siguiente;
                    siguiente.prev = anterior;
                    }
            count--;
            }
            if (rdo==null){
                    return null;
            }else{
                    return rdo.data;
            }
    //COSTE O(1)
}


    public T remove(T elem) {
```

```
  //Elimina un elemento concreto de la lista

  Node<T> rdo     = null;

  if(!contains(elem)){

  }else{
          rdo = findDevuelveNodo(elem);
          Node<T> anterior = rdo.prev;
          Node<T> siguiente = rdo.next;

          anterior.next = siguiente;
          siguiente.prev = anterior;

          count--;
  }

  if (rdo==null)
          return null;
  else
          return rdo.data;

  //COSTE O(1)
  };

  public T first() {
  //Da acceso al primer elemento de la lista
        if (isEmpty())
            return null;
        else return first.data;
  }

  public T last() {
  //Da acceso al ultimo elemento de la lista
        if (isEmpty())
            return null;
        else return first.prev.data;
  }

  public boolean contains(T elem) {
  //Determina si la lista contiene un elemento concreto
if (isEmpty())
    return false;

Node<T> current = first.next; // Empieza con el segundo elemento

while ((current != first) && !elem.equals(current.data))
```

```
        current = current.next;
   return elem.equals(current.data);
}

     private Node<T> findDevuelveNodo (T elem) {
     //Determina si la lista contiene un elemento concreto,
     //y develve su referencia, null en caso de que no este

             Node<T> aux = null;

             if(contains(elem)){
                 aux = first;
                 boolean encontrado = false;
                 while((!aux.data.equals(last()))&&(!encontrado)){
                     if(aux.data.equals(elem)){
                             encontrado=true;
                     }else{
                             aux=aux.next;
                     }
                 }
             }
             //Si aux es el ultimo justo habra salido del while
             //y lo devolvera bien
             return aux;
             //COSTE O(n)
     }

     public T find(T elem) {
     //Determina si la lista contiene un elemento concreto, y develve su
     //referencia, null en caso de que no esta
    Node<T> nodo = findDevuelveNodo(elem);
    if (nodo == null) return null;
    else return nodo.data;

     //COSTE O(n)
     }

     public boolean isEmpty()
     //Determina si la lista esta vacia
     { return first == null;};

     public int size()
     //Determina el numero de elementos de la lista
     { return count;};

     /** Return an iterator to the stack that iterates through the items . */
     public Iterator<T> iterator() { return new ListIterator(this); }
```

```
// an iterator, doesn't implement remove() since it's optional
private class ListIterator implements Iterator<T> {

     private Node<T> actual;
     private int indice;

     public ListIterator(DoubleLinkedList l){
             actual = l.first.prev;
             indice = 0;
     }

     public boolean hasNext() {
             if(indice==count){
                     return false;
             }else{
                     return true;
             }
     }

     public T next() {
         try{
             if(hasNext())
             {
                 T rdo = actual.next.data;
                 actual = actual.next;
                 indice++;
                 return rdo;
             }
             else
             {
                 throw new NoHayNextException();
             }
         }
         catch(NoHayNextException ex)
         {
                 //System.out.println("No hay siguiente.");
                 T rdo = actual.data;
                 return rdo;
         }
     }

     public void remove() {
             actual.prev.next = actual.next;
             actual.next = actual.prev;
     }
}
```

```
                public void visualizarNodos() {
                        System.out.println(this.toString());
                }


                @Override
                public String toString() {
                    String result = new String();
                    Iterator<T> it = iterator();
                    while (it.hasNext()) {
                            T elem = it.next();
                            result = result + "[" + elem.toString() + "] \n";
                    }
                    return "SimpleLinkedList " + result + "]";
                }
}
```

5.1.2 Clase IndexedListADT

```
/**
 * IndexedListADT defines the interface to an indexed list collection. Elements
 * are referenced by contiguous numeric indexes.
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/13/08
 */

package listasSimples;

public interface IndexedListADT<T> extends ListADT<T>
{
   /**
    * Inserts the specified element at the specified index.
    *
    private int indice;   //el indice del array donde introduciremos el elemento
    private T elemento;    //the element to be inserted into the array
    */
   public void add (int index, T element);

   /**
    * Sets the element at the specified index.
    *
    * @param index   the index into the array to which the element is to be set
    * @param element the element to be set into the list
    */
   public void set (int index, T element);
```

```
   /**
    * Adds the specified element to the rear of this list.
    *
    * @param element  the element to be added to the rear of the list
    */
   public void add (T element);

   /**
    * Returns a reference to the element at the specified index.
    *
    * @param index  the index to which the reference is to be retrieved from
    * @return       the element at the specified index
    */
   public T get (int index);

   /**
    * Returns the index of the specified element.
    *
    * @param element  the element for the index is to be retrieved
    * @return         the integer index for this element
    */
   public int indexOf (T element);

   /**  Removes and returns the element at the specified index. */
   public T remove (int index);
}
```

5.1.3 Clase ListADT

```
package listasSimples;

import java.util.Iterator;

public interface ListADT<T> {

public void setDescr(String nom);
// Actualiza el nombre de la lista

public String getDescr();
// Devuelve el nombre de la lista

public T removeFirst();
//Elimina el primer elemento de la lista

public T removeLast();
//Elimina el ultimo elemento de la lista
```

```
public T remove(T elem);
//Elimina un elemento concreto de la lista

public T first();
//Da acceso al primer elemento de la lista

public T last();
//Da acceso al ultimo elemento de la lista

public boolean contains(T elem);
//Determina si la lista contiene un elemento concreto

public T find(T elem);
//Determina si la lista contiene un elemento concreto, y
//develve su referencia, null en caso de que no esta

public boolean isEmpty();
//Determina si la lista esta vacia

public int size();
//Determina el numero de elementos de la lista

public Iterator<T> iterator();

}
```

5.1.4 Clase NoHayNextException

```
package listasSimples;

public class NoHayNextException extends Exception {

        public NoHayNextException() {}

        public NoHayNextException(String message){
            super(message);
          }
}
```

5.1.5 Clase NoInt

```
package listasSimples;
public class NoInt extends Exception {
        public NoInt() {}
        public NoInt(String message){
            super(message);
          }
}
```

5.1.6 Clase Node

```
package listasSimples;
	public class Node<T> {
		public T data;			// dato del nodo
		public Node<T> next;	// puntero al siguiente nodo de la lista,
					//apunta al primero si es el ultimo
		public Node<T> prev;	// puntero al anterior nodo de la lista,
					//apunta al ultimo si es primero

		// ------------------------------------------------------------

		public Node(T dd)		// constructor
		{
			data = dd;
			next = null;
			prev = null;
		}
	}
```

5.1.7 Clase OrderedDoubleLinkedList

```
package listasSimples;

public class OrderedDoubleLinkedList<T> extends DoubleLinkedList<T> implements OrderedListADT<T> {

	public void add(T elem) {
		if(!this.contains(elem)){
	    Node<T>elemact=new Node<T>(elem);
	    String elem_act=elemact.data.toString();
	    if(esInteger(elem_act)){
		int pos=buscarPosAntInt(elem);
		if (pos==0){
		addToFront(elem);
		}
		else{
		    Node<T> nuevo,anterior;
		    nuevo=new Node<T>(elem);
		    anterior=first;
		    int cont=1;
		    while(cont!=pos){
			    anterior=anterior.next;
			    cont++;
		    }
		    nuevo.next = anterior.next;
		    nuevo.prev = anterior;
		    anterior.next = nuevo;
		    nuevo.next.prev = nuevo;
		}
```

```
    }
    else{
        int pos=buscarPosAnt(elem);

            if (pos==0){
                    addToFront(elem);
            }else{
                    Node<T> nuevo,anterior;
                    nuevo=new Node<T>(elem);
                    anterior=first;
                    int cont=1;
                    while(cont!=pos){
                            anterior=anterior.next;
                            cont++;
                    }
                    nuevo.next = anterior.next;
                    nuevo.prev = anterior;
                    anterior.next = nuevo;
                    nuevo.next.prev = nuevo;
            }
    }
    count++;
        }
}

private int buscarPosAnt(T elem){
    int cont=0;
    boolean salir=false;
    Node<T> anterior=first;
    Node<T> actual=new Node <T> (elem);
    if(first!=null)
        while(!(salir||(anterior==first&&cont>0))){
            Comparable c_act = (Comparable)actual.data.toString();
            Comparable c_ant = (Comparable)anterior.data.toString();
            if(c_act.compareTo(c_ant)==-1){
                salir=true;
            }
            else{
                anterior=anterior.next;
                cont++;
            }
        }
    return cont;
}

private void addToFront(T elem) {
    Node <T> nuevo= new Node<T>(elem);
```

```
    if (first==null){
            //Lista vacia
            first=nuevo;
            first.prev = nuevo;
            first.next = nuevo;
    }else{
            Node<T> anterior, siguiente;
            anterior = first.prev;
            siguiente = first;
            first = nuevo;
            siguiente.prev = nuevo;
            anterior.next = nuevo;
            nuevo.prev = anterior;
            nuevo.next = siguiente;
    }
}

private boolean esInteger( String input )  {
    try
    {
       Integer.parseInt( input );
       return true;
    }
    catch( Exception ex)
    {
       return false;
    }
 }


private int buscarPosAntInt(T elem){
    int cont=0;
    boolean salir=false;
    Node<T> anterior=first;
    Node<T> actual=new Node <T> (elem);
    if(first!=null)
        while(!(salir||(anterior==first&&cont>0))){
            String c_act_str=actual.data.toString();
            int c_act= Integer.parseInt(c_act_str);
            String c_ant_str=anterior.data.toString();
            int c_ant= Integer.parseInt(c_ant_str);
            if(c_act<(c_ant)){
                    salir=true;
            }
            else{
                        anterior=anterior.next;
                        cont++;
```

```
                        }
                }
            return cont;
        }

}
```

5.1.8 Clase OrderedListADT

```
package listasSimples;

public interface OrderedListADT<T>  extends ListADT<T> {

        public void add(T elem);
        // Añade un elemento a la lista (en el lugar de orden que le corresponde)
}
```

5.1.9 Clase Persona

```
package listasSimples;

public class Persona implements Comparable<Persona> {

        // atributos
        private String name;
    private String dni;

        public Persona(String pName, String pDni) { // Constructora
                name = pName;
                dni = pDni;
        }

        public String getName() { return name; }

        public void setName(String name) { this.name = name; }

        public String getDni() { return dni; }

        public void setDni(String dni) { this.dni = dni; }

        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                Persona other = (Persona) obj;
```

```
		if (dni == null) {
			if (other.dni != null)
				return false;
		} else if (!dni.equals(other.dni))
			return false;
		return true;
	}

	public int compareTo(Persona arg0) {
		return name.compareToIgnoreCase(arg0.name);
	}

	public String toString() {
		return name + " " + dni;
	}

} // end Persona
```

5.1.10 Clase UnorderedDoubleLinkedList

```
package listasSimples;

public class UnorderedDoubleLinkedList<T> extends DoubleLinkedList<T> implements UnorderedListADT<T> {

	public void addToFront(T elem) {
		if(!this.contains(elem)){
			Node <T> nuevo= new Node<T>(elem);
			if (first==null){
				//Lista vacia
				first=nuevo;
				first.prev = nuevo;
				first.next = nuevo;
			}else{
				Node<T> anterior, siguiente;
				anterior = first.prev;
				siguiente = first;
				first = nuevo;
				nuevo.next = siguiente;
				siguiente.prev = nuevo;
				nuevo.prev = anterior;
				anterior.next=nuevo;
			}
			count++;
		}
	}

	public void addToRear(T elem) {
		if(!this.contains(elem)){
```

```
                Node<T> nuevo = new Node<T>(elem);
                if (first==null){
                        //Lista vacia
                        first=nuevo;
                        first.prev = nuevo;
                        first.next = nuevo;
                }else{
                        Node<T> anterior;
                        anterior = first.prev;
                        nuevo.prev = anterior;
                        anterior.next = nuevo;
                        nuevo.next = first;
                        first.prev = nuevo;
                }
                count++;
        }
}

public void addAfter(T elem, T target) {
        if(!this.contains(elem)){
                int pos=tomarPos(target);
                if (pos==-1){
                        System.out.println("No se ha encontrado el elemento.");
                }
                else{
                        Node<T> nuevo,actual;
                        nuevo=new Node<T>(elem);
                        actual=first;
                        int cont=1;
                        while(cont!=pos){
                                actual=actual.next;
                                cont++;
                        }
                        nuevo.next=actual.next;
                        actual.next=nuevo;
                }
                count++;
        }
}

public int tomarPos(T elem){
        Node<T> actual=first;
        int cont=0;
        boolean salir=false, enc=false;
        while(!(enc||salir||actual==null)){
                if(actual.data.equals(elem)){
                        enc=true;
```

```
                }
                        else{
                                actual=actual.next;
                                cont++;
                                if(actual.equals(first)){
                                    actual=null;
                                    salir=true;
                                }
                        }
                }
                if (actual==null){
                        return -1;
                }
                else{
                return cont;
                }
        }
}
```

5.1.11 Clase UnorderedListADT

```
package listasSimples;

public interface UnorderedListADT<T> extends ListADT<T> {

        public void addToFront(T elem);
        // añade un elemento al comienzo

        public void addToRear(T elem);
        // añade un elemento al final

        public void addAfter(T elem, T target);
        // Añade elem detras de otro elemento concreto, target,  que ya se encuentra en la lista
}
```

5.1.12 Clase ListaActores

Esta es la clase que hemos modificado del primer laboratorio para que funcione con la nueva estructura de listas ligadas.

```
package lab1;
import java.util.Iterator;
import listasSimples.OrderedDoubleLinkedList;

public class ListaActores {
        //Atributos
        private static ListaActores milistadeactores = new ListaActores();
```

```
private OrderedDoubleLinkedList<Actor> milista;

//Constructora
public ListaActores(){
        milista = new OrderedDoubleLinkedList<Actor>();
}

//Getters y Setters
public static ListaActores getListaActores(){
        return milistadeactores;
}

public OrderedDoubleLinkedList<Actor> getMilista(){
        return this.milista;
}

public Iterator<Actor> getIterador(){
        return this.getMilista().iterator();
}

public int getTamano(){
        return this.getMilista().size();
}


//Otros Metodos
        public boolean esta(Actor pActor){
                return this.getMilista().contains(pActor);
        }

        public void anadirActor(Actor pActor){
                try{
                        if(esta(pActor)){
                                System.out.println("El actor ya se encuentra en la lista");
                        }else{
                                getMilista().add(pActor);
                        }
                }catch(NullPointerException e){
                        System.out.println("El actor que desea eliminar no existe");
                }
        }

public void eliminarActor(Actor pActor){
        try{
                if(esta(pActor)){
                        getMilista().remove(pActor);
                }else{
```

```
                        System.out.println("El actor no se encuentra en la lista");
                }
        }catch(NullPointerException e){
                System.out.println("El actor que desea eliminar no existe");
        }
}

public Actor buscarActor(Actor pActor){
        try{
                if(esta(pActor)){
                        return this.getMilista().find(pActor);
                }else{
                        return null;
                }
        }catch(NullPointerException e){
                System.out.println("El actor que intentas buscar no existe");
                return null;
        }
}

public void resetear(){
        getListaActores().getMilista().setFirst(null);
}

public void imprimir(){
        getListaActores().milista.visualizarNodos();
}

/*COMO LA LISTA YA ESTA ORDENADA NO NECESITA ESTOS MÉTODOS.
 *
 *
 * public void ordenarLista(){
        Actor[] milistaordenada = this.convertirHashArray();
        ordenacionPorBurbuja(milistaordenada);
        JOptionPane.showMessageDialog(null, "El resultado se muestra por consola");
        for(int i = 0; i<milistaordenada.length; i++)
                System.out.println(i+": "+milistaordenada[i].getNombre());
}

public Actor[] convertirHashArray(){
        Actor[] miArrayDeActores = new Actor[this.getMilista().size()];
        int i = 0;
        Iterator<String> itr = this.getIterador();
        while (itr.hasNext()){
                miArrayDeActores[i] = this.getMilista().get(itr.next());
                i++;
        }
```

```
		return miArrayDeActores;
	}

	public void ordenacionPorBurbuja(Actor[] tabla) {
		int out, in;
		for (out = tabla.length - 1; out > 0; out--)
			for (in = 0; in < out; in++)
				if ( tabla[in].compareTo(tabla[in + 1]) > 0 )
					swap(tabla, in, in + 1);
	}

	private void swap(Actor[] tabla, int a, int b) {
		Actor aux = tabla[a];
		tabla[a] = tabla[b];
		tabla[b] = aux;
	}*/
}
```

5.2 Pruebas

En este apartado mostraremos las pruebas unitarias realizadas, clasificadas por clases.

5.2.1 Clase PruebaDoubleLinkedList

```
package pruebasListasSimples;

import java.util.Iterator;
import listasSimples.UnorderedDoubleLinkedList;
public class PruebaDoubleLinkedList {

	public static void visualizarNodos(UnorderedDoubleLinkedList<Integer> l) {
		Iterator<Integer> it = l.iterator();
		System.out.println();
		while (it.hasNext()) {
			Integer num = it.next();
			System.out.println(num);
		}
	}


	public static void main(String[] args)  {

		UnorderedDoubleLinkedList<Integer> l =new UnorderedDoubleLinkedList<Integer>();
		l.addToRear(1);
		l.addToRear(3);
		l.addToRear(6);
```

```
		l.addToRear(7);
		l.addToRear(9);
		l.addToRear(0);
		l.addToRear(20);
		l.addToFront(8);
		l.remove(new Integer(7));


		System.out.print(" Lista ..............");
		visualizarNodos(l);
		System.out.println(" Num elementos: " + l.size());


		System.out.println("Prueba Find ..............");
		System.out.println("9? " + l.find(9));
		System.out.println("0? " + l.find(0));
		System.out.println("7? " + l.find(7));

		int aux=l.tomarPos(1);
		System.out.println("Este valor tiene que dar 1 y da " + aux);
		aux=l.tomarPos(2);
		System.out.println("El valor no se encuentra y tiene que dar -1, y da " + aux);
		aux=l.tomarPos(3);
		System.out.println("Este valor tiene que dar 2 y da " + aux);

		l.addAfter(2, 1);
	System.out.print(" Lista ..............");
		visualizarNodos(l);
	}

	public void testTomarPos(){
		UnorderedDoubleLinkedList<Integer> l =new UnorderedDoubleLinkedList<Integer>();
		l.addToRear(1);
		l.addToRear(3);
		l.addToRear(6);
		l.addToRear(7);
		int aux=l.tomarPos(1);
		System.out.println("este es el valor de " + aux);
	}
}
```

5.2.2 Clase PruebaOrderedDoubleLinkedList

```
package pruebasListasSimples;

import listasSimples.OrderedDoubleLinkedList;
import listasSimples.Persona;
```

```
public class PruebaOrderedDoubleLinkedList {

		public static void main(String[] args)  {

			OrderedDoubleLinkedList<Integer> l = new OrderedDoubleLinkedList<Integer>();
			l.add(1);
			l.add(3);
			l.add(6);
			l.add(7);
			l.add(9);
			l.add(0);
			l.add(20);
			l.remove(new Integer(7));


			System.out.print(" Lista ..............");
			l.visualizarNodos();
			System.out.println(" Num elementos: " + l.size());


			System.out.println("Prueba Find ..............");
			System.out.println("20? " + l.find(20));
			System.out.println("9? " + l.find(9));
			System.out.println("9? " + l.find(9));
			System.out.println("0? " + l.find(0));
			System.out.println("7? " + l.find(7));



			OrderedDoubleLinkedList<Persona> l2 = new OrderedDoubleLinkedList<Persona>();
			l2.add(new Persona("jon", "1111"));
			l2.add(new Persona("ana", "7777"));
			l2.add(new Persona("amaia", "3333"));
			l2.add(new Persona("unai", "8888"));
			l2.add(new Persona("pedro", "2222"));
			l2.add(new Persona("olatz", "5555"));

			l2.remove(new Persona("", "8888"));


			System.out.print(" Lista ..............");
			l2.visualizarNodos();
			System.out.println(" Num elementos: " + l2.size());


			System.out.println("Prueba Find ..............");
			System.out.println("2222? " + l2.find(new Persona("", "2222")));
			System.out.println("5555? " + l2.find(new Persona("", "5555")));
```

```
                        System.out.println("7777? " + l2.find(new Persona("", "7777")));
                        System.out.println("8888? " + l2.find(new Persona("", "8888")));
        }
}
```

5.2.3 Clase PruebaListaActores

```
package pruebasListasSimples;
import lab1.Actor;
import lab1.ListaActores;

public class PruebaListaActores {

        public static void main(String args[]){
                ListaActores l = new ListaActores();
                Actor a1 = new Actor("a");
                Actor a2 = new Actor("b");
                Actor a3 = new Actor("c");
                Actor a4 = new Actor("d");

                l.anadirActor(a1);
                l.anadirActor(a4);
                l.anadirActor(a4);
                l.anadirActor(a2);
                l.anadirActor(a3);
                l.eliminarActor(a1);


                System.out.println("Lista....");
                l.imprimir();
                System.out.println("N�mero de elementos: "+l.getTamano());


                System.out.println("Prueba Find ..............");
                System.out.println("b? " + l.buscarActor(a2).getNombre());
                System.out.println("d? " + l.buscarActor(a4).getNombre());
                System.out.println("null? "+l.buscarActor(a1));

        }
}
```

6 Conclusiones

En este segundo laboratorio hemos aprendido a trabajar con la estructura de datos lista doblemente ligada circular. Hemos tenido que diseñar algoritmos para acceder a los diferentes datos que alberga la estructura y también a ordenar los datos. Además de esto hemos aplicado esta estructura al laboratorio anterior.